Designing Mediation for Context-Aware Applications

Jennifer Mankoff, and Anind Dey

IRB-TR-03-005

February, 2003

Submitted to ACM Transactions on Computer-Human Interaction Journal, special issue on Sensing-Based Interactions

Intel Research Berkeley

Designing Mediation for Context-Aware Applications

Anind Dey, and Jennifer Mankoff

IRB-TR-03-005

February, 2003

Submitted to ACM Transactions on Computer-Human Interaction Journal, special issue on Sensing-Based Interactions

Intel Research Berkeley

Designing mediation for context-aware applications

ANIND K. DEY
Intel Research, Berkeley
and
JENNIFER MANKOFF
UC Berkeley

Many context-aware services make the assumption that the context they use is completely accurate. However, in reality, both sensed and interpreted context is often ambiguous. A challenge facing the development of realistic and deployable context-aware services, therefore, is the ability to handle ambiguous context. In this paper, we describe an architecture that supports the building of context-aware services that assume context is ambiguous and allows for mediation of ambiguity by mobile users in aware environments. We discuss design heuristics that arise from supporting mediation over space and time. We illustrate the use of our architecture and the design heuristics and evaluate it through three example context-aware services, a word predictor system, an In/Out Board, and a reminder tool.

Categories and Subject Descriptors: H5.2 [**Information Interfaces and Presentation**]: User Interfaces – *Graphical user interfaces*; *Interaction styles*; D.2.11 [**Software Architectures**]: Software Engineering - *Domain-specific architectures*;
General Terms: Context-aware Computing, Ambiguity, Aware Environments, Ubiquitous computing, Mediation, Error handling
Additional Key Words and Phrases:

---

1. INTRODUCTION

A characteristic of an aware, sensor-rich environment is that it senses and reacts to *context*, information sensed about the environment's mobile occupants and their activities, by providing context-aware services that facilitate the occupants in their everyday actions. Researchers have been building tools and architectures to facilitate the creation of these context-aware services by providing ways to more easily acquire, represent and distribute raw sensed data and inferred data [**19**]. Our experience shows that though sensing is becoming more cost-effective and ubiquitous, the interpretation of sensed data as context is still imperfect and will likely remain so for some time. A challenge facing the development of *realistic* and *deployable* context-aware services, therefore, is the ability to handle imperfect, or *ambiguous*, context.

Authors' addresses: Anind K. Dey, Intel Research Berkeley, Intel Corporation, Berkeley, CA 94704; Jennifer Mankoff, EECS Department, UC-Berkeley, Berkeley, CA 94720-1770.

A framework proposed by Bellotti *et al*. for addressing the challenges in designing the communication aspects of sensing-based systems [**1**]. In this framework, 5 issues are introduced: how to address individual devices in a sensor-rich environment, how to know that the system is attending to the user, how to take action, how to know that the system has taken the correct action, and how to avoid mistakes. In this paper, we attempt to address the issues of providing *feedback* to support users in knowing that the system is attending to them and in knowing what the system has done and providing the ability to *disambiguate* sensed input to avoid the system taking incorrect actions.

In our work, we have taken a systems approach to addressing these issues. We are looking at the architectural support required to support feedback and disambiguation. In doing so, we have built a number of applications that use ambiguous context. Our experiences enabled us to develop some design heuristics for future sensing-based systems and for supporting sensing-based interactions. The contribution that this paper presents are the design guidelines and a runtime architecture that supports programmers in the development of multi-user, interactive, distributed applications that use ambiguous data and the discussion of the guidelines and architecture in the context of three applications we have built.

1.1 Mediation

Researchers in aware environments have used techniques from the artificial intelligence (AI) community, including Bayesian networks and neural networks [**8**,**20**], to deal with imperfect context. However, the techniques cannot remove all the ambiguity in the sensed data, leaving it up to the aware environment programmer and occupants to deal with. To alleviate this problem, we propose to leverage techniques for reducing the ambiguity and involve end users in removing any remaining ambiguity, through a process called *mediation* [**17**].

In graphical user interface (GUI) design, mediation refers to the dialogue between the user and computer that resolves questions about how the user's input should be interpreted in the presence of ambiguity. A common example of mediation in recognition-based GUIs is the *n*-best list. Ambiguity arises when a recognizer is uncertain as to the current interpretation of the user's input, as defined by the user's intent. An application can choose to ignore the ambiguity and just take some action (*e.g.* act on the most likely choice), or can use mediation techniques to ask the user about her actual intent.

Ambiguous context, from an aware environment, can produce errors similar to those in recognition-based interfaces. In the case of ambiguous context, there are additional

challenges that arise from the inherent mobility of humans in aware environments. Specifically, since users are likely to be mobile in an aware environment, the interactions necessary to provide feedback in alerting them to possible context errors (from sensing or the interpretation of sensed information) and allow for the disambiguation and smooth correction of those errors must occur over some time frame and over some physical space. We discuss some of the new architectural mechanisms and heuristics that come into play, since designing correction strategies over time and space involves more than just an architecture that supports error mediation.

1.2 System Architecture

In previous work, we presented an architecture for the development of context-aware services that assumed context to be unambiguous [11]. We also developed an architecture to support the mediation of ambiguity in recognition-based GUI interfaces [**17**]. Building on this past work, we developed support for the *additional* architectural requirements (justified in the next section) that arise as a result of requesting highly mobile users to mediate ambiguous context in *distributed, interactive,* sensing environments [10]. Our architecture supports the building of applications that allow humans in an aware environment to detect errors in sensed information about them and their intentions and correct those errors in a variety of ways. In particular it supports:

- Timely delivery and update of ambiguous events across an interactive distributed system;
- Delayed storage of context once ambiguity is resolved;
- Delivery of ambiguous context to multiple applications that may or may not be able to support mediation;
- Pre-emption of mediation by another application or component;
- Applications or services in requesting that another application or service mediate; and,
- Distributed feedback about ambiguity to users in an aware environment.

Our runtime architecture addresses these issues and supports our goal of building more realistic context-aware applications that can handle ambiguous data through mediation.

1.3 Design Guidelines

During the course of our research in sensor-based interactions, we have built a number of applications that use ambiguous context and support users in mediating this ambiguity. In doing so, we have generated an initial set of design heuristics.

- Applications should provide *redundant mediation techniques* to support more natural and smooth interactions;
- Applications should provide facilities for providing input and output that are *distributed both in space and time* to support input and feedback for mobile users; and,
- Interpretations of ambiguous context should have *carefully chosen defaults* to minimize user mediation.

We will discuss these guidelines in the context of three applications that we have built.

1.4 Overview of Paper

We begin by presenting a motivating example used to illustrate the requirements of mediation in a context-aware setting. In the next section, we present brief overviews of previous work that we have extended: the Context Toolkit, an infrastructure to support the rapid development of context-aware services, which has assumed perfect context sensing in the past; and OOPS (Organized Option Pruning System), an architecture for the mediation of errors in recognition-based interfaces. We show how they were combined to deal with ambiguous context, and describe additional architectural mechanisms that were developed for the requirements unique to mediation of context in a distributed setting. We describe the design guidelines and show how the architecture supports exploration with them. We then present three case studies that illustrate how the architecture supports mediation of ambiguous context and how the design guidelines can be applied. The three applications are a word prediction system for the disabled (our motivating example), an In/Out Board and a context-aware reminder system. We conclude the paper with related work and a discussion of further challenges in mediating interactions in context-aware applications.

2. MOTIVATING EXAMPLE

We have developed three applications as demonstrations of our architecture. One in particular, a context-aware communication system, the Communicator, will be used to illustrate key points throughout this paper, and we introduce it here.

The Communicator is designed for people with motor and speech impairments. For these people, exemplified by Stephen Hawking, computers can provide a way to communicate with the world and increase both independence and freedom. Many people with severe motor impairments can control only a single switch, triggered by a muscle that is less spastic or paralyzed than others. This switch is used to scan through screen elements, such as the keys of a soft keyboard. Input of this sort is very slow and is often enhanced by word prediction.

The Communicator, shown in Figure 1, is based on a word predictor that attempts to predict what word a user is typing from the letters that have been typed so far. The non-speaking individual uses the interface shown Figure 1a. The keyboard layout shown was chosen for optimal efficiency for scanning input [16]. Text is displayed to the (abled) communication partner at top, reversed for easy readability by someone facing the user, across a flat display and in a separate interface (Figure 1b). Word predictors are very inaccurate, and because of this, they usually display a list of possible predictions that the user scans through for the correct choice, often to no avail. Word prediction is especially difficult to use for spoken communication because the speed of conversational speech often reaches 120 words per minute (wpm) or more, while users of word prediction rarely go above 10 wpm.

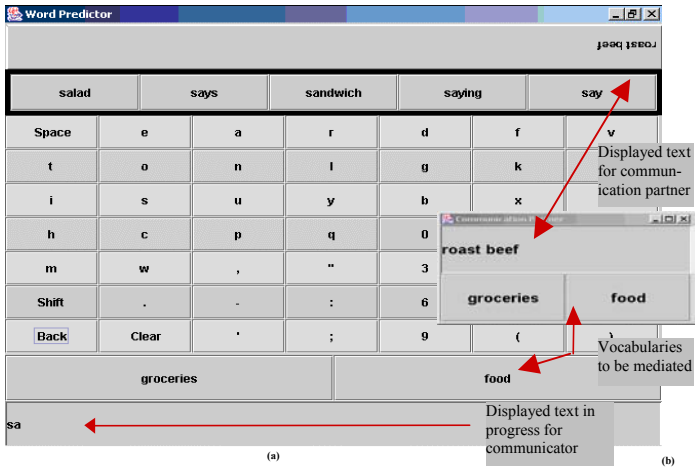


Figure 1 (a) Communicator (back) and partner (right, front) interfaces.

The goal of the Communicator is to facilitate conversational speech through improved word prediction. We augment word prediction by using a third party intelligent system, the Remembrance Agent [21], to select conversation topics, or *vocabularies,* based on contextual information including recent words used, a history of the user's previous conversations tagged with location and time information, the current time and date and the user's current location. These vocabularies help to limit the set of predicted words to those that are more relevant and thus improve prediction. For example, when in a bank, words such as "finance" and "money" should be given priority over other similar words. This has been shown to be effective for predicting URLs in Netscape™ and Internet Explorer™ and, in theory, for non-speaking individuals [15,18]. Our goal was to build an application to support context-aware word prediction for non-speaking individuals.

Unfortunately, it is hard to accurately predict the topic of a user's conversation, and because of this, the vocabulary selection process is ambiguous. We experimented with several mediation strategies, ranging from simply and automatically selecting the top choice vocabulary without user intervention to stopping the conversation in order to ask the user which vocabulary is correct.

3. ARCHITECTURE REQUIREMENTS

The focus of this paper is on support for programmers to build realistic context-aware applications. This includes providing design guidelines for building applications (presented in a later section) and addressing the architectural issues needed to support mediation of ambiguous input. On the architecture side, there must exist a system that is able to capture context and deliver it to interested consumers, and there must be mediation techniques for managing ambiguity. These issues were dealt with in our previous work. In the following subsections we discuss the interesting additional challenges that arise from mediating ambiguous context, all of which are supported by the architecture presented here.

3.1 Context Acquisition and Ambiguity

One common characteristic of context-aware applications is the use of sensors to collect data. In the Communicator, location and time information is used to help improve word prediction. A user's location can be sensed using Active Badges, radar, video cameras or GPS units. All of these sensors have some degree of ambiguity in the data they sense. A vision system that is targeted to identify and locate users based on the color of the clothing they wear will produce inaccurate results if multiple users are wearing the same color of clothing. The ambiguity problem is made worse when applications derive implicit higher-level context from sensor data. For example, an application may infer that a meeting is occurring when a number of users move into the same room in a given time interval. However, there may be other explanations for this phenomenon, including random behavior, lunchtime or the workday has started and multiple people are arriving at their desk. Even with the use of sophisticated Bayesian networks or other AI techniques, low- and high-level inferences are not always correct, resulting in ambiguity.

3.2 Distribution

Most context-aware applications are distributed across multiple computing devices. Applications or system components that are interested in context (often called *subscribers*) are running on devices that are remote from components that are gathering context. The component gathering the context may not be the component that mediates it, since it may not have an interface. In the Communicator, the user's interface and the

communication partner's interface are running on separate devices. It is important to minimize the number and duration of network calls in an interactive distributed system, and thus, to only send the information absolutely needed for mediation to only those components that are performing the mediation.

3.3 Storage

Because context-aware systems are often distributed and asynchronous, and because sensor data may be used by multiple applications, it is beneficial to store data being gathered by sensors. The Communicator takes advantage of stored information by accessing past conversations that match the user's current location and time. Storing context data allows applications that were not running at the time the data was collected to access and use this historical data. When that data is ambiguous, several versions must be saved, making the storage requirements prohibitive. Interesting issues to address are when should we store data (before or after ambiguity is resolved) and what should we store (ambiguous or unambiguous context).

3.4 Multiple Subscription Options

In many context-aware systems, multiple subscribers are interested in a single piece of sensed input. An interesting issue is how to allow individual components to "opt in" to ambiguous context while allowing others to "opt out". Some components may wish to deal with ambiguity while others may not. For example, non-interactive components such as a data logging system may not have any way to interact with users and therefore may not support mediation. Other components, like the Communicator interface, may only wish to receive unambiguous data. In the same manner, a logging system might wish to only record data that is certain. A second issue to deal with is allowing components to deal with ambiguous data while not requiring them to perform mediation. Later in the paper, we will discuss a word predictor widget in the Communicator that has this same property.

3.5 Pre-emption of Mediation

In our system, multiple, completely unrelated components may subscribe to the same ambiguous data source. Both Communicator interfaces have the ability to mediate ambiguous vocabularies, for example. An important concern to resolve is what to do when these components start to mediate that data at the same point in time.

3.6 Forced Mediation

There are cases where a subscriber does not wish to mediate ambiguous data itself, but may still wish to exert some control over the timing of when another subscriber completes mediation. One way of doing this is allowing it to request immediate

mediation by others. In the Communicator, when a conversation ends, a component responsible for managing past conversations wants to store this conversation in an appropriate vocabulary. This component does not have an interface, so it requests that the application mediate the possible vocabularies.

3.7 Feedback

When distributed sensors collect context about a user, a context-aware system needs to be able to provide feedback about the ambiguous context to her, particularly when the consequences are important to her. In a typical aware environment, users are mobile and may interact with multiple devices throughout their interaction with the environment. For this reason, the architecture needs to support the use of remote feedback, providing feedback (visual or aural, in practice) on a device that may be remote from both the subscribing component and the sensing component. Take the previous example of a user's motion being monitored by a video camera to provide identity and location-based services. As the user moves down a hallway, a device on the wall may display a window or use synthesized speech to indicate who the video camera system thinks the user is. This device is neither a subscriber of the context nor the context sensor, but simply has the ability to provide useful feedback to users about the state of the system. We will present an implemented example of feedback in our discussion of the reminder application.

In the next section, we will discuss the architecture we designed and implemented to deal with these requirements.

4. MEDIATING AMBIGUOUS CONTEXT

We built support for mediation of imperfectly sensed context by extending an existing toolkit, the Context Toolkit [9]. The Context Toolkit is a software toolkit for building context-aware services that support mobile users in aware environments, using context it assumes to be perfect. The toolkit make it easy to add the use of context or implicit input to existing applications that do not use context. There are two basic building blocks that are relevant to this discussion: context widgets and context interpreters. Figure 2 shows the relationship between context components and applications.

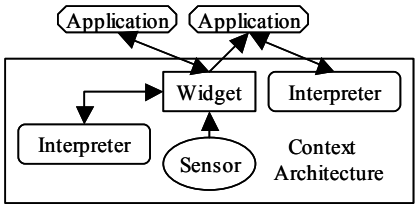


Figure 2 Context Toolkit components: arrows indicate data flow.

Context widgets, presented elsewhere [9], are based on an analogy to GUI widgets. GUI widgets encapsulate information about a single interactive element, and provide abstractions that allow applications developers to easily communicate with them. Context widgets encapsulate information about a single piece of context, such as location or activity, for example. Some context that the existing toolkit supports include motion, temperature, identity (iButtons), volume level, and many others. Developers can also easily add additional sensors to the infrastructure. Applications may use any combination of these sensors.

Context widgets are responsible for collecting contextual information about the environment and its occupants. They provide a uniform interface to components or applications that use the context, hiding the details of the underlying context-sensing mechanism(s). These widgets allow the use of heterogeneous sensors that sense redundant input, regardless of whether that input is implicit or explicit. Widgets maintain a persistent record of all the context they sense. They allow applications and other widgets to both query and subscribe to the context information they maintain.

A *context interpreter* is used to abstract or interpret context. For example, a context widget may provide location context in the form of latitude and longitude, but an application may require the location in the form of a street name. A context interpreter may be used to provide this abstraction. A more complex interpreter may take context from many widgets in a conference room to infer that a meeting is taking place. Both interpreters and widgets are sources of ambiguous data.

Context components are intended to be persistent, running 24 hours a day, 7 days a week. They are instantiated and executed independently of each other in separate threads and on separate computing devices. The Context Toolkit makes the distribution of the context architecture transparent to context-aware applications, handling all communications between applications and components.

4.1 Modifications for Mediation

In order to explain how we met the requirements given in the previous section, we must first introduce the basic abstractions we use to support mediation. We chose to base our work on the abstractions first presented in the OOPS toolkit [17], a GUI toolkit that provides support for building interfaces that make use of recognizers (*e.g.* speech, gestures) that interpret user input. Like sensing context, recognition is ambiguous and OOPS provides support for tracking and mediating uncertainty. We chose OOPS because it explicitly supports mediation of single-user, single application, non-distributed, ambiguous desktop input, a restricted version of our problem.

OOPS provides an internal model of recognized input, based on the concept of hierarchical events [21], that allows separation of mediation from recognition and from the application. As we will see, this is a key abstraction that we will use in the extended Context Toolkit. This model encapsulates information about ambiguity and the relationships between input and interpretations of that input that are produced by recognizers in a graph (See Figure 3). The graph keeps track of source events, and their interpretations (which are produced by one or more recognizers).

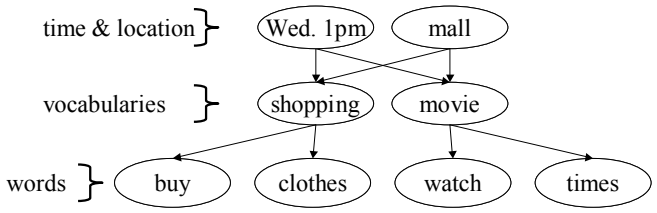


Figure 3 An event graph representing predicted words from context.

Like OOPS, our toolkit automatically identifies ambiguity in the graph and intervenes between widgets and interpreters and the application by passing the directed graph to a *mediator*. A mediator is an interface element that allows the user and computer to communicate about ambiguity. Mediators generally fall into two major categories. Choice mediators give the user a choice of possible interpretations of her input. Repetition mediators support the user in repeating her input, usually in an alternate and less error-prone modality. OOPS provides a library including standard mediators from these categories.

OOPS automatically handles the task of routing input to mediators when it is ambiguous, and informing recognizers and the application about the correct result when ambiguity is resolved. This separation of mediation from recognition and from the application means that the basic structure of an application and its interface does not need to be modified in order to add to or change how recognition or mediation is done. Additionally, the directed graph provides a consistent internal model that makes it possible to build mediators that are completely independent of recognizers.

A mediator displays a portion of the graph to the user. Based on the user's response, the mediator *accepts* or *rejects* events (*i.e.* keeps correct interpretations or removes incorrect interpretations) in the graph. Once the ambiguity is resolved (all events in the graph are accepted or rejected), the toolkit allows processing of the input to continue as normal. Figure 4 shows the resulting changes. The gray boxes indicate components that have been added to the Context Toolkit architecture illustrated in Figure 2 to support mediation of ambiguous context.

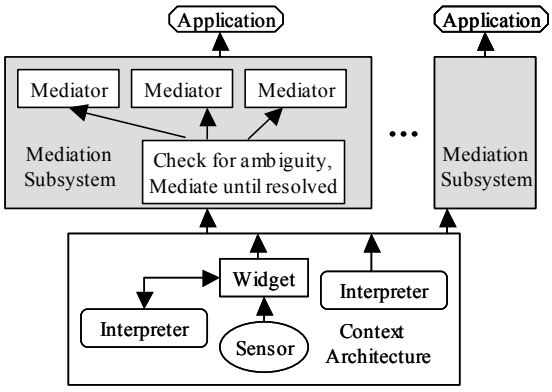


Figure 4 The architecture for the extended Context Toolkit. Everything in the gray boxes is new.

4.2 Example

Before discussing the additional changes necessary to support the requirements listed above, we illustrate the use of ambiguous hierarchical events in the Context Toolkit with an example. In the Communicator system, time and location information is used to choose relevant vocabularies. An intelligent recognition system provides the most likely vocabularies and then these are interpreted into the words the user is most likely to be typing. The set of vocabularies and the set of words are stored as sets of alternatives with associated confidences (a fairly common representation). Each of these alternatives becomes an ambiguous event in our system. The result is a directed graph, like that shown in Figure 3.

Often only one path through this graph is correct from the user's perspective (*e.g.* mall & Wednesday → shopping → clothes). We call this situation *ambiguous* and mediation is used to resolve the ambiguity. In particular, a mediator will display feedback about one or more interpretations to the user, who will then select one or repeat her input.

Now suppose that an application subscribes to this data. All three of the applications we present later make use of location data. Subscribers to location data may then:

1) Wait until all the ambiguity has been resolved before taking any action on a location update; or,

2) Take action on the ambiguous data by:

a) Asking for a user to help mediate the data;

b) Picking one of the alternatives (usually the one with the highest confidence) and acting on it; or

c) Performing some higher-level inference (such as the word a user is typing) with its own ambiguity. This increases the depth and complexity of the event graph.

4.3 Modifications for New Requirements

The previous subsections described the basic abstractions used to support mediation: widgets, interpreters, applications, mediators and the event graph. We now explain the additional architectural mechanisms needed to support the unique problems faced by mediation of ambiguous context in a distributed, multi-user setting, introduced above.

4.3.1 Distribution

The original OOPS toolkit was designed to support interactive mediation in non-distributed GUI applications. It always passed a pointer to the entire event graph to mediators.

In order to support appropriate response times in the distributed environment of the Context Toolkit, only those portions of the event graph that have subscribers are passed across the network. Otherwise, each time a new event is added or an existing event is accepted or rejected, every component interested in the ambiguous context would have to be notified. In a distributed system, this would impede our ability to deliver context in a timely fashion, as is required to provide feedback and action on context.

No one component contains the entire graph being used to represent ambiguity of a particular piece of context. The graph is, instead, distributed across multiple components (widgets and interpreters) and copies of particular graph levels are provided to applications, as needed. Each event or element in the graph has a list of its source events (parent(s)) and its interpretations (children). Rather than having the lists contain full representations of the sources and interpretations, the lists instead contain event proxies. An event proxy consists of an event id, the status (accepted, rejected or undetermined) of the event and communication information (hostname, port number, component name) for the component that produced the event and contains its full representation. Because components mostly care about the status of their sources and interpretations, the proxies allow components to operate as if they had local access to the entire graph and to request information about parts of the event graph that they do not have locally.

4.3.2 Storage

As described above, storage of context data is a useful feature of a context-aware architecture. However, when context is ambiguous, it is not immediately obvious what should be stored and when. One option is to store all data, regardless of whether it is ambiguous or not. This option provides a history of user mediation and system ambiguity that could be leveraged at some later time to create user models and improve recognizers' abilities to produce interpretations. We chose to implement a less complex option: By default, every widget stores only unambiguous data. Another dimension of storage relates

to when data is stored. Since we are storing unambiguous data only, we store context data only after it has been mediated.

The reason for our choice is two-fold: the storage policy is easier to deal with from an access standpoint and we gain the benefits offered by knowledge of ambiguity during the mediation process, just not at some arbitrary time after mediation (when the record of ambiguity has been discarded). In any case, it would be relatively simple to modify the architecture to support the first option as a default.

4.3.3 Multiple Subscription Options

Because multiple components may be interested in the same piece of context, and only some may be interested in ambiguous data, components need a way of specifying whether they want to handle ambiguous data. In our architecture, they simply set a Boolean flag to specify this.

Components that accept ambiguous data are not required to perform mediation. They can take any action they wish with the unmediated data. Components that accept unambiguous data also are not required to perform mediation, but they must wait until another component does (or force mediation, as described below) before they receive the data.

In either case, when a component successfully mediates data, other components interested in the data are notified. The architecture keeps track of all the recipients of the ambiguous data and updates them. As well, it keeps track of any components waiting for unambiguous versions of the data and passes the mediated data to them. Finally it notifies the components that produced the ambiguous data that can use the data to improve their ability to produce new data.

4.3.4 Pre-Emption of Mediation

Because multiple components may subscribe to the same ambiguous data, mediation may actually occur simultaneously in these components. If multiple components are mediating at once, the first one to succeed “interrupts” the others and updates them with the mediated data. This is handled automatically by the architecture when the successful mediator accepts or rejects data. The architecture notifies any other recipients about the change in status. Each recipient determines if the updated data is currently being mediated locally. If so, it informs the relevant mediators that they have been pre-empted and should stop mediating. Past work did not handle mediation in multiple distributed components. Other strategies for handling simultaneous mediation are discussed in the future work section.

4.3.5 Forced Mediation

In cases where a subscriber of ambiguous context is unable to or does not want to perform mediation, it can request that another component perform it. The subscriber simply passes the set of ambiguous events it wants mediated to a remote component and asks that remote component to perform mediation. If the remote component is unable to do so, it notifies the requesting component. Otherwise, it performs mediation and updates the status of these events, allowing the requesting component to take action.

4.3.5 Feedback

Since context data may be gathered at locations remote from where the active application is executing and at times remote from when the user is interacting with the active application, there is a need for distributed feedback services that are separate from applications. This allows mediation to occur in the user's location where the input was sensed or where they are currently located, independent of the location of the actual application. To support distributed feedback, we have extended context widgets to support feedback and actuation via output services. Output services are quite generic and can range from sending a message to a user to rendering some output to a screen to modifying the environment. Some existing output services render messages as speech; send email or text messages to arbitrary display devices; and control appliances such as lights and televisions. Any application or component can request that an output service be executed, allowing any component to provide feedback to a user.

5. DESIGN GUIDELINES: EXPLORING DISTRIBUTED MEDIATION IN PRACTICE

In the previous sections, we provided motivation for supporting mediation of ambiguous context and providing feedback about what the system knows about the user and the environment. We also presented modifications to an existing architecture to support these features. In the remainder of the paper, we will present design guidelines for building sensor-based applications and demonstrate how the architectural solutions provided by the modified Context Toolkit are put into practice in more realistic sensing-based interactions. We will describe three examples that not only describe the specifics of applying the modified Context Toolkit, but a demonstration of important heuristics that go beyond what an architecture can provide and which come up in designing distributed mediation when mobility is involved. We have gathered an initial set of heuristics described below:

- Applications should provide *redundant mediation techniques* to support more natural and smooth interactions;

- Applications should provide facilities for providing input and output that are *distributed both in space and time* to support input and feedback for mobile users; and,
- Interpretations of ambiguous context should have *carefully chosen defaults* to minimize user mediation.

5.1 Providing Redundant Mediation Techniques

One of the attractive features of context-aware computing is the promise that it will allow humans to carry out their everyday tasks without having to provide additional explicit cues to some computational service. Our experience shows, however, that the more implicit the gathering of context, the more likely it is to be in error. In designing mediation techniques for correcting context, a variety of redundant techniques should be provided simultaneously. This redundant set not only provides a choice on the form of user input and system feedback, but also the relative positioning and accessibility to the user should be carefully thought out to provide a smooth transition from most implicit (and presumably least obtrusive) to the most explicit [22].

With respect to the architectural modifications described above, support for distribution, pre-emption of mediation and feedback are necessary to apply and explore this design heuristic. Distribution of the event hierarchy allows multiple mediators access for mediating ambiguous context. Pre-emption of mediation deals with issues specific to the case where multiple mediators are active. Feedback supports mediators in disclosing information to end users about the information being mediated.

5.2 Spatio-temporal Relationship of Input and Output

Some input must be sensed before any interpretation and subsequent mediation can occur. Because we are assuming user mobility, this means that the spatial relationship of initial input sensors must mesh with the temporal constraints to interpret that sensed input before providing initial feedback to the user. Should the user determine that some mediation is necessary, that feedback needs to be located within physical range of the sensing technologies used to mediate the context and the space through which the user is moving. Mediating interactions should occur along the natural path that the user would take. In some cases, this might require duplicate sensing technologies to take into account different initial directions in which a user may be walking. In addition, the mediation techniques may need to have a carefully calculated timeout period, after which mediation is assumed not to happen.

Again, support for distribution of the event hierarchy is important for exploring this heuristic as it supports mediation occurring over multiple spaces. Pre-emption of

mediation supports multiple mediators working simultaneously in different locations that may be along the user's expected path, offering the user multiple opportunities to mediate data. Support for feedback is important for providing information to the user as she moves throughout her environment.

5.3 Effective Use of Defaults

Sometimes the most effective and pleasurable interactions are ones that do not have to happen. Prudent choices of default interpretations can result in no additional correction being required from the user. These defaults could either provide some default action or provide no action, based on the situation. For example, in a situation involving highly ambiguous context, it may be best to do nothing by default and only take action if the user indicates the correct interpretation through mediation.

While the architecture modifications do not directly influence the use of this design heuristic, there is some impact. Support for multiple subscription types and storage allow components that produced the ambiguous context to improve their ability to produce new context and inferences, or, in other words, to produce better defaults.

6. CASE STUDIES

In this section, we describe the building of 3 context-aware applications, the Communicator, an In/Out Board and a reminder system. The first application was built from scratch, using both ambiguous and unambiguous data sources. The other two were modifications of existing applications to include ambiguous sensors and mediation. These applications validate our architecture and illustrate how our design heuristics can be applied to real systems.

6.1 The Communicator

We introduced the Communicator system as our motivating example. Its interface is shown in Figure 1. Here we will describe the physical setup of the application, the interaction supported, the system architecture and how the design heuristics were applied in building this application.

6.1.1 Physical Setup

The Communicator runs on a laptop computer, attached to a wheelchair. A GPS unit is mounted on the wheelchair and connected to the computer. As the user moves from location to location in a downtown city environment, the Communicator suggests appropriate vocabularies based on the current location and time, shown near the bottom of the interface. The user can select a vocabulary at any time during a conversation to help the system predict appropriate words. As the user starts entering characters with the scanning interface, the system predicts potential word completions shown near the top of

the interface, using words from the system-suggested vocabularies or the user-selected vocabulary, to support him in maintaining a conversation. The user can select any of the suggested words or continue to type individual characters. If the user is speaking with a companion using a partner device, the companion can also select the appropriate vocabulary on behalf of the user.

6.1.2 Mediation

We experimented with four different strategies for mediating ambiguous vocabularies. The first simply accepts the vocabulary with the highest probability without user input (equivalent to no mediation at all). The second (see Figure 5a) displays the choices similarly to words, and allows the user to ignore them. The last two require the user to choose a vocabulary at different points in the conversation (Figure 5b). The third requires a choice when a new conversation starts and new ambiguous vocabularies are suggested. The fourth displays the choices, but only requires that the user choose one when a conversation has ended. The mediated vocabulary name is used to append the current conversation to the appropriate vocabulary file, which then improves future vocabulary/word prediction. These approaches demonstrate a range of methods whose appropriateness is dependent on recognizer accuracy. The architecture easily supports this type of experimentation by allowing programmers to easily swap mediators.

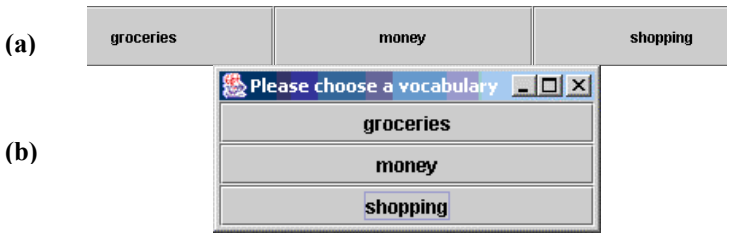


Figure 5 Screenshots of mediators (a) choice mediator for words or vocabularies and (b) required mediator for vocabularies.

6.1.3 Architecture

We will now discuss how the architecture facilitated the building of the Communicator application. Figure 6 shows the architecture described below.

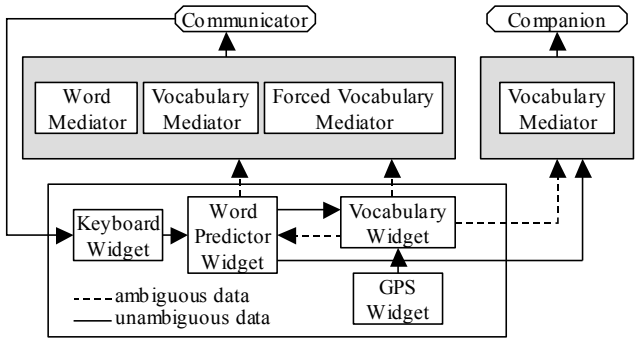


Figure 6 Architecture for the Communicator System

The Communicator makes direct use of data from three widgets: a soft keyboard, a word predictor and a vocabulary selector. The keyboard widget produces unambiguous data and simply lets other components know what the user is typing. The word predictor widget produces ambiguous data and uses the current context to predict what word the user is trying to type. It uses a unigram, frequency-based method common in simple word predictors, as well as a history of recent words. It subscribes to the keyboard to get the current prefix (the letters of the current word that have been typed so far). As each letter is typed, it suggests the most likely completions. The word predictor also uses weighted vocabularies to make its predictions. It subscribes to the vocabulary widget to get a list of ambiguous, probable vocabularies and uses the probability associated with each suggested vocabulary to weight the words from that vocabulary. As described earlier, the vocabulary widget uses the Remembrance Agent [22] to suggest relevant, yet ambiguous vocabularies for the current conversation.

If the person the user is communicating with also has a display available, a companion application can be run. This application presents an interface (see Figure 1), showing the unambiguous words selected by the user and the current set of ambiguous vocabularies.

This application uses two unambiguous widgets (GPS and keyboard), and two widgets that generate ambiguous data, one based on a third party recognizer (vocabulary), and one based on an in-house recognizer (word). Unlike typical context-aware systems, ambiguity in our systems is retained, and, in some cases, displayed to the user.

Ambiguous information generated in our system includes potential vocabularies and potential words. The architecture allows a component to mediate ambiguous context, use it as is, or use it once something else has mediated it. All three cases exist in this system. The application mediates both ambiguous words and vocabularies. The word predictor uses ambiguous vocabularies. The vocabulary widget uses unambiguous words after the user has mediated them. The word mediator is graphical and it displays ambiguous words as buttons in a horizontal list, shown *in situ* near the bottom of Figure 1a. A word may be selected by the user or ignored. The mediator replaces all the displayed words whenever it receives new words from the word predictor.

We will now describe how the architecture works from a system perspective. When all of the widgets and user interfaces are started, the word predictor generates an initial set of guesses of likely words, based on an "empty" prefix from the keyboard widget.

The source event (the empty prefix) is sent to the word predictor for interpretation and the interpretations (predicted words) are passed to a handler in the user interface (UI), which immediately routes them to the word mediator for display because they are ambiguous. The user may select one, in which case, the mediator accepts that word and rejects all of the others. The toolkit then proceeds to notify the interpretations' and source event's producers (word predictor and keyboard widgets, respectively), and all the recipients. The word predictor adds the accepted word to a "recent words" list used to enhance prediction. The Communicator UI and the companion application's UI (Figure 1) display the word to the user and companion.

If the user types a letter with the soft keyboard, that letter is passed to the Communicator UI (which displays it at the bottom) and to the word predictor. The word predictor uses that and all subsequent letters as sources of its predictions and once again the user may resolve the predictions by selecting a word.

Meanwhile, the vocabulary widget attempts to find relevant vocabularies every time the user enters a new word or the user changes location. These ambiguous vocabularies are received by the word predictor widget, which then predicts new words (see Figure 7). The potential vocabularies are displayed by a vocabulary mediator in both the Communicator UI and the companion's UI. If either person selects a vocabulary, the architecture notifies the other mediator that it has been pre-empted. When using the fourth vocabulary mediation strategy, the vocabulary widget forces mediation to request selection of a vocabulary at the end of a conversation (signaled by a long break in keyboard use). The architecture passes this request on to each subscriber to see if it can perform mediation.

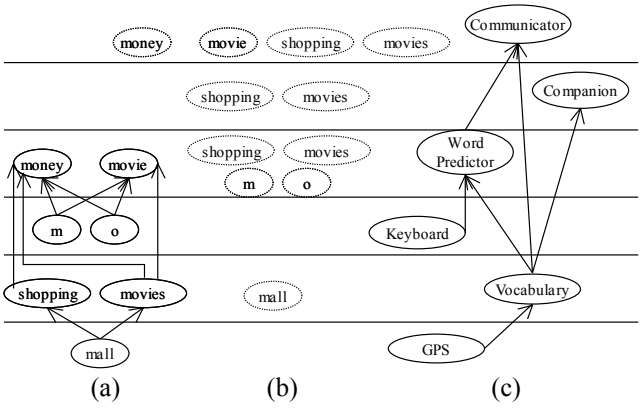


Figure 7 (a) Sample event graph and distribution across components (c). Events exist both in the components that created them and the (b) components they were sent to (*e.g.* the "money" word event exists in the Word Predictor Widget and the Communicator interface).

The Communicator receives the request and creates the dialog box mediator shown in Figure 5b. The user interacts with the mediator and selects a vocabulary. The event hierarchy is updated, and the vocabulary widget is notified that an event it created has been accepted. The widget writes the conversation out to disk in the appropriate vocabulary file.

This application demonstrates two important features of the architecture. First, it shows that it supports experimentation with mediation by making it trivial to swap mediators in and out. Adding or replacing a mediator only requires two lines of code. Second, it shows it is not difficult to build a compelling and realistic application. The main Communicator application consists of only 435 lines of code, the majority dealing with GUI issues. Only 19 lines are for mediation and 30 deal with context acquisition.

6.1.3 Design Issues

We will now discuss the Communicator with respect to the design heuristics we have proposed.

Redundant mediation techniques: The Communicator supports mediation of both vocabularies and words. As stated earlier, for vocabularies, we experimented with a variety of mediation techniques. Depending on the technique, the system automatically chooses a vocabulary, the user can choose a vocabulary at any time or the user is forced to choose a vocabulary at a particular time. Additionally, a companion can choose the correct vocabulary for the user on a separate device. By investigating a range of techniques, we will determine which one(s) is the most appropriate in a given setting. For the mediation of words, users can either select one of the words suggested or can continue typing causing a new set of suggested words to appear. Providing this range of mediation techniques offers flexibility to the end user.

Spatio-temporal relationship of input and output: In the Communicator system, interaction is distributed in space when the Communicator interface and the Companion interface are being used simultaneously. Vocabulary and word mediation may distributed over time depending on the mediation technique being used, ranging from system-decided to user-decided input.

Appropriate use of defaults: Providing appropriate defaults and reducing keyboard input is the main purpose of this application. The different mediators we experimented with supported different defaults. The mediator that automatically chose the most likely vocabulary is most appropriate when there is little ambiguity, whereas the mediator that does not require the user to select a vocabulary is most appropriate when there is a lot of ambiguity.

6.2 In/Out Board

The first application we modified is the In Out Board [9], an application that displays the current in/out status for occupants of a home. In the original system, an unambiguous location widget informed the application when a user entered or left the building. Users indicated their status by docking a Java iButton®.

6.2.1 Physical Setup

Occupants of the home are detected when they arrive and leave through the front door, and their state on the In-Out Board is updated accordingly. Figure 8(a) shows the front door area of our instrumented room, taken from the living room. In the photographs, we can see a small anteroom with a front door and a coat rack. The anteroom opens up into the living room, where there is a key rack and a small table for holding mail – all typical artifacts near a front door. To this, we have added two ceiling-mounted motion detectors (one inside the house and one outside), a display, a microphone, speakers, a keyboard and a dock beside the key rack.

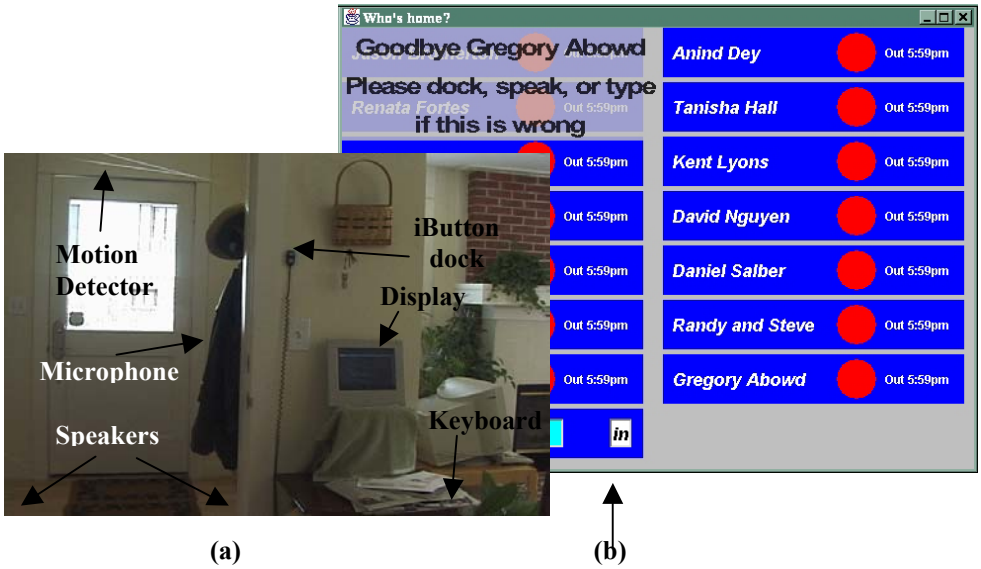


Figure 8 (a) Photograph of In-Out Board physical setup; **(b)** In-Out Board with transparent graphical feedback.

6.2.2 Mediation

When an individual enters the home, the motion detectors detect his presence. The current time, the order in which the motion detectors were set of and historical information about people entering and leaving the home is used to infer who the likely individual is and whether he is entering or leaving. While the original version of the application would simply have updated the in or out status of the individual at this point, the modified In-Out Board includes mediators that give the user a chance to help when there is ambiguity.

First, a mediator speaks the most likely inference using synthesized speech. For example, it might say “Hello Jane Doe” or “Goodbye Fred Smith”. In addition, the wall display shows a transparent graphical overlay (see Figure 8(b)) indicating the current state and how the user can correct it if it is wrong: speak, dock or type.

If the inference is correct, the individual can simply continue on as usual. After a timeout period of 20 seconds, the mediator will select the top choice and the In-Out Board will be informed and update its display with this new information. If the inference is incorrect, the direction (exiting or entering) and/or identity may be wrong. The individual can correct the inference using a combination of speech input, docking with an iButton, and keyboard input on the display. These input techniques plus the motion detectors range from being completely implicit to extremely explicit. Each of these techniques can be used either alone, or in concert with one of the other techniques. There is no pre-defined order to their use. Since some of these inputs may themselves be ambiguous, the mediator responds to the refinements by resetting a timer, and providing additional feedback indicating how the new information is assimilated, by speaking and displaying the change.

Changes can continue to be made indefinitely, however, if the user makes not change for a pre-defined amount of time, mediation is considered to be complete. As the timer counts down, the transparent display slowly fades away. When the timer expires, the current choice is selected and the service updates the wall display with the corrected input. The timeout for this interaction is 20 seconds.

For example, the user can say things like “No”, “No, I’m entering/exiting”, “No, it’s John Doe”, or “No, it’s John Doe and I’m entering/exiting”. The speech recognition system is not assumed to be 100% accurate, so the system again indicates its updated understanding of the current situation via synthesized speech and the transparent overlay.

Alternatively, an occupant can dock her iButton. An iButton is a button that contains a unique id that the system uses to determine the identity of the occupant. The system then makes an informed guess based on historical information as to whether the user is coming or going. The user can further refine this using any of the techniques described if it is wrong.

Finally, the occupant can use the keyboard to correct the input. By typing his name and a new state, the system’s understanding of the current situation is updated.

6.2.3 Architecture

We will now discuss how the architecture facilitated this service. The following figure (Figure 9) shows the block diagram of the components in the system.

The application was originally written using the unmodified Context Toolkit. Input captured via context widgets detected presence using an iButton. We modified the application to use additional widgets from the original Context Toolkit: motion detectors, speech recognition, and keyboard widgets. All of the widgets were modified to be able to generate ambiguous as well as unambiguous context information.

Rather than requiring explicit action from the user to determine in/out status, the motion detector-based widget uses an interpreter to interpret motion information into user identity and direction. The interpreter uses historical information collected about occupants of the house, in particular, the times at which each occupant has entered and left the house on each day of the week. This information is combined with the time when the motion detectors were fired and the order in which they were fired. A nearest-neighbor algorithm is then used to infer identity and direction of the occupant. The speech recognition-based widget uses a pre-defined grammar to determine identity and direction.

Because the original system did not support ambiguity, it ignored the fact that docking an iButton® merely provides information about user presence and not about user arrival or departure from a room. The motion detector widget not only introduces this ambiguity about user state, but, in an attempt to require less explicit user action, also introduces additional ambiguity about the user's identity.

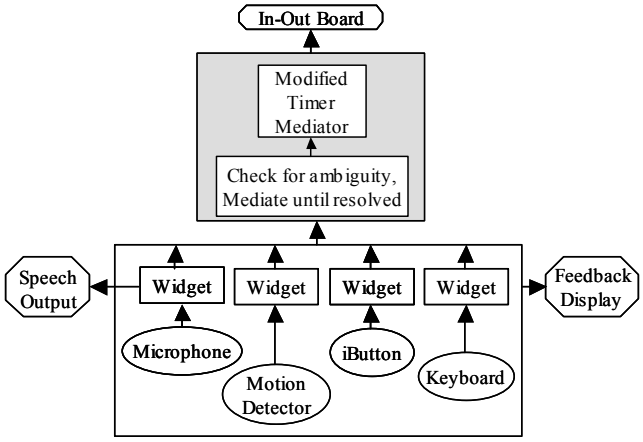


Figure 9 Architecture diagram for In-Out Board. There are 4 widgets providing context, two of which have output services for feedback: speech output and visual feedback display.

When any of the widgets capture input, they produce not only their best guess as to the current situation, but also other likely alternatives, creating an ambiguous event graph. The wall display has subscribed to unambiguous context information and is not interested in ambiguous information.

When ambiguous information arrives, it is intercepted by a mediator that resolves it with the user's help so that it can be sent to the application in its unambiguous form. The mediator uses this ambiguous information to mediate (accept and reject) or refine alternatives in the graph. The entire ambiguous graph is not held by any one component. Instead, it is distributed among the four context widgets and the mediator. Each component can obtain access to the entire graph, but it is not necessary in this service.

The mediator displays the current best guess to the user (Figure 8b), and allows her to correct it in a variety of modalities ranging from lightweight to heavyweight, including speech, docking her iButton®, and typing at a keyboard.

The mediator is an instance of a temporal mediator, which creates a timer to create temporal boundaries on this interaction. The timer is reset if additional input is sensed before it runs out. As the mediator collects input from the user and updates the graph to reflect the most likely alternative, it provides feedback to the user. It does this in two ways. The first method is to use a generic output service provided by the Context Toolkit. This service uses IBM ViaVoice™ to produce synthesized speech to provide feedback to the user. The second method is application-specific and is the transparent graphical overlay on the wall display shown in Figure 8. The transparent overlay indicates what the most likely interpretation of the user's status is and what the user can do to change their status: e.g. "Hello Anind Dey. Please dock, type, or speak if this is wrong." As the timer counts down, the overlay becomes more transparent and fades away.

When all the ambiguity has been resolved in the event graph and the timer has expired, the overlay will be faded completely and the correct unambiguous input is delivered to the wall display and the display updates itself with the new status of the occupant. Also, the input is delivered back to the interpreter so it has access to the updated historical information to improve its ability to infer identity and direction.

The only direct modification of the In-Out Board application was to install the mediator and widgets described above. A total of 22 lines were changed or added. 14 were minor substitutions where references were changed from the unambiguous widget to the ambiguous one and three were new library imports. Two new class variables were created to hold pointers to the mediator and three lines of code were added to create the mediator and pass it one piece of necessary information about the application, a pointer to its user interface.

The newly modified widgets used by the In/Out Board are reusable and one of them is, in fact, used by the next application as well. The mediator we use is an extension of a mediator from our library of mediators, modified to display application-specific text.

6.2.4 Design Issues

In this section, we will further investigate the design heuristics, introduced in a previous section, that arose during the development of this service.

Redundant mediation techniques: On the input side, in an attempt to provide a smooth transition from implicit to explicit input techniques, we chose motion detectors, speech, docking and typing. In order to enter or leave the house, users must pass through the doorway, so motion detectors are an obvious choice to detect this activity. Often users will have their hands full, so speech recognition is added as a form of more explicit, hands-free input. IButton docking provides an explicit input mechanism that is useful if the environment is noisy. Finally, keyboard input is provided as an additional explicit mechanism and to support the on-the-fly addition of new occupants and visitors.

A valid question to ask is why not use sensors that can be more accurately interpreted. Unfortunately in practice, due to both social and technological issues, there are few sensors that are both reliable and appropriate. As long as there is a chance that the sensors may make a mistake, we need to provide users with techniques for correcting these mistakes. None of the sensors we chose are foolproof either, but the combination of all the sensors and the ability to correct errors before applications take action is a satisfactory alternative.

Spatio-temporal relationship of input and output: The next design decision is where to place the input sensors and the rendering of the output to address the spatio-temporal characteristics of the physical space being used. There are "natural" interaction places in this space, where the user is likely to pause: the door, the coat rack, the key rack and the mail table. The input sensors were placed in these locations: motion sensors on the door, microphone in the coat rack, iButton dock beside the key rack and keyboard in a drawer in the mail table. The microphone being used is not high quality and requires the user to be quite close to the microphone when speaking. Therefore the microphone is placed in the coat rack where the user is likely to be leaning into when hanging up their coat. A user's iButton is carried on the user's key chain, so the dock is placed next to the key rack. The speakers for output are placed between the two interaction areas to allow it to be heard throughout the interaction space. The display is placed above the mail table so it will be visible to individuals in the living room and provide visual feedback to occupants using the iButton dock and keyboard.

To support feedback, synthesized speech is used both to mirror the speech recognition input and to provide an output mechanism that is accessible (*i.e.* audible) to the user

throughout the entire interaction space. Visual output for feedback is provided in the case of a noisy environment and for action as a persistent record of the occupancy state.

Effective use of defaults: Another design issue is what defaults to provide to minimize required user effort. We use initial feedback to indicate to the user that there is ambiguity in the interpreted input. Then, we leave it up to the user to decide whether to mediate or not. The default is set to the most likely interpretation, as returned by the interpreter. Through the use of the timeout, the user is not forced to confirm correct input and can carry out his normal activities without interruption. This is to support the idea that the least effort should be expended for the most likely action. The length of the timeout, 20 seconds, was chosen to allow enough time for a user to move through the interaction space, while being short enough to minimize between-user interactions.

Other design issues: We added the ability to deal with ambiguous context, in an attempt to make these types of applications more realistic. Part of addressing this realism is dealing with situations that may not occur in a prototype research environment, but do occur in the real world. An example of this situation is the existence of visitors, or people not known to the service. To deal with visitors, we assume that they are friendly to the system, a safe assumption in the home setting. That means they are willing to perform minimal activity to help keep the system in a valid state. When a visitor enters the home, the service provides feedback (obviously incorrect) about who it thinks this person is. The visitor can either just say "No" to remove all possible alternatives from the ambiguity graph and cause no change to the display, or can type in her name and state using the keyboard and add herself to the display.

6.3 CybreMinder

The second application we modified is CybreMinder [12], a situation-aware reminder system. The original application subscribes to every widget that is running and allows the user to create reminders for themselves or someone else triggered by any situation, or combination of events, that these widgets might generate. For example, a user might set up a reminder to go to a meeting when at least three other people are present in the meeting room at the right time. Delivery of reminders is performed whenever the current context appears to match the triggers specified by the user. The application assumes that the reminder has been successfully delivered and acted upon. The purpose of this system is to trigger and deliver reminders at more appropriate opportunities than is currently possible.

6.3.1 Physical Setup

Various locations (building entrances and offices) in two buildings have been instrumented with iButton docks to determine the location of building occupants. With each dock is a computer screen on which reminder messages can be displayed. Figure 10 shows an example installation.

Figure 10 Reminder display with iButton dock.

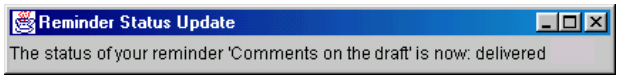
6.3.2 Mediation

Users can create situation-aware reminders on any networked device. We will illustrate the interaction through the use of a concrete example. Jane and David are working on a paper for UIST. Jane sent out a draft of the paper and is waiting for comments. She creates a reminder message for David to drop off his comments. She sets the situation in which to deliver the reminder to be when David enters the building.

Figure 11 Reminder message delivered in appropriate situation.

When David enters the building (sensed by the appropriate iButton dock), the reminder is delivered to him on the closest display that also beeps to get his attention (Figure 11). Just because the reminder was delivered to David, does not mean that he will complete the action detailed in the reminder. In fact, the most likely occurrence in this setting is that a reminder will be put off until a later time. The default status of this reminder is set to "still pending". This means that the reminder will be delivered again, the next time David enters the building. However, if David does enter Jane's office within a pre-defined amount of time (10 minutes), as indicated by him docking his iButton, the system changes the previous incorrect reminder status to "completed" and the application is informed that the reminder has been delivered (Figure 12a). Of course,

if he was just stopping by to say hello, he can dock again to return this back to "still pending" (Figure 12b).

(a) 

(b) 

Figure 12 Graphical reminder status: (a) "delivered" and (b) "pending".

6.3.3 Architecture

We will now discuss how the architecture facilitated this service. The following figure (Figure 13) shows the block diagram of the components in this system.

Input is captured via context widgets that detect presence, using iButtons as the input-sensing mechanism. When the information from these widgets matches a situation for which there is a reminder, the reminder is delivered to a display closest to the recipient's location. The reminder is displayed using an output service that the widgets provide.

Initially, input in this system was treated as unambiguous in [12]. Using the combination of the Context Toolkit and OOPS, we have added the ability to deal with ambiguity. Now, a pair of ambiguous events is created by the widget, one indicating that the reminder is still pending and one indicating that the reminder has been completed.

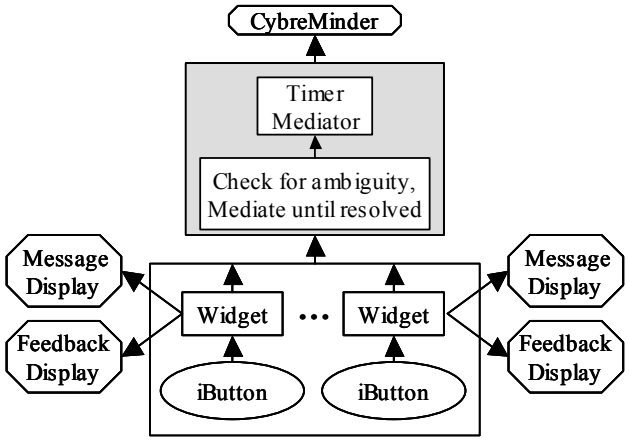


Figure 13 Architecture diagram for reminder service. There is one widget for each "interesting" location in the building. Each widget has two services, one for displaying a reminder and one for providing feedback about the reminder's status.

As with the In-Out Board, the reminder system has subscribed for unambiguous context, so when the ambiguous events arrive, they are delivered to the mediator for the service. We modified the reminder delivery mechanism by adding this mediator to remove the assumption that a reminder has been successfully delivered and acted upon. The mediator gives the user the opportunity to reject a reminder within a certain time after its delivery. This indicates that the reminder should be re-delivered the next time the

current context matches the trigger. If the user does not reject it, the system proceeds to change its status to 'delivered' just as it would have done immediately in the original application.

When the mediator receives this input, it creates a timer to enforce temporal boundaries on this interaction. The timer has a timeout value of 10 minutes for this service. If the user does not address and complete the reminder, the timer times out and the most likely event is chosen, that of keeping the reminder status as "still pending". If the user does address the reminder, he can dock his iButton to indicate this. This docking event is delivered to the mediator which swaps which reminder status is most likely.

Feedback is provided to the user via an audio cue and on the display that is closest to the user's current location (Figure 10) using another output service provided by the local widget. Each additional dock swaps the reminder status as well and produces feedback for the user. Timely delivery of docking events from the widget to the mediator and back to the widget to provide feedback is essential for providing the user with timely feedback. When the timer expires, the most likely reminder status is passed to the service that updates the reminder accordingly.

The original CybreMinder application was only modified to subscribe to all iButton® widgets so the application would be notified when a user docked to mediate a reminder and to install a reusable temporal mediator that selects and updates the most likely interpretation after a timeout (as in the case of the In/Out Board). The mediator makes use of remote widget feedback services to display feedback about the reminder status. It is an extension of a timer mediator modified to display application-specific messages. The application modifications required the addition of 3 library imports and the modification or addition of 27 lines of code.

6.3.4 Design Issues

In this section, we will further investigate some of the interesting design issues that arose during the development of this service.

Redundant mediation techniques: In this service, input is provided to the system using iButtons and docks, which comprise the location system in our research building. The existing infrastructure was leveraged for this service, both for simplicity of development and to leverage off of user's knowledge about the system. Additionally, users can explicitly correct the status of a reminder using a separate interface from the comfort of their desk. Output comes in the form of a simple audio cue, a beep, to get the user's attention, and a visual cue that indicates the current reminder status. Speech input was not used in this setting because it was a more public space than the home.

Spatio-temporal relationship of input and output: The choice of locations for input and output was again guided by the space in which the service was deployed. Natural locations for sensing input and providing feedback were the entrances to rooms where users would naturally stop to dock anyway. If a different location system were used, the locations chosen might differ slightly. Entrances to offices would still be appropriate, as they are natural stopping places where users knock on a door. But in a conference room, the chairs where users site may be a better choice.

Effective use of defaults: In this service, as opposed to the previous one, the default interpretation of user input is that no action was taken, and that the reminder is still pending. This default was chosen because the implicit user input received (a user entering the building) only causes a reminder to be delivered, and does not provide any indication as to whether the reminder has been addressed and completed. There is really no sensor or group of sensors that will enable us to accurately determine when a reminder has been addressed. We must rely on our users to indicate this. The interpretation that a reminder is still pending is the most likely interpretation and therefore it was made the default, requiring the least user action to be accepted. The timeout for accepting the input was chosen in a similar fashion as the first service, long enough to give the user an opportunity to address the reminder, while short enough to minimize overlap between individual interactions (reminders).

Other design issues: When designing this service, we chose to address the ambiguity only at the level of whether the reminder was dealt with or not. This was done in order to make the design simpler for demonstration purposes. The underlying context that is used to determine when a message should be delivered will also have ambiguous alternatives that may need mediation. It should not be hard to see how we could combine the type of service we demonstrated with the In-Out Board with this reminder service, to make this possible.

6.4 In Summary

We built on application from scratch and modified two existing applications. Between them, they demonstrate all the required features of the architecture. They use (and reuse) a number of ambiguity-generating widgets. The first application was built from scratch and very little of its code was dedicated to dealing with mediation or context acquisition. The other two applications required minor modifications to deal with ambiguity. All three applications involve distributed event hierarchies and use reusable mediators to resolve ambiguity. The applications help to illustrate the validity of the underlying architecture and to demonstrate how the design heuristics can be analyzed and applied in real settings.

7. RELATED WORK

Over the past several years, there have been a number of research efforts aimed at creating a ubiquitous computing environment, as described by Weiser [24]. We divide these efforts into three categories: aware environments, architectures to support context-aware services, and relevant context-aware services.

An *aware environment* is an environment that can automatically or *implicitly* sense information or context about its occupants and itself and can take action on this context. In the Reactive Room project, a room used for video conferencing was made aware of the context of both users and objects in the room for the purpose of relieving the user of the burden of controlling the objects [7]. For example, when a figure is placed underneath a document camera, the resulting image is displayed on a local monitor as well as on remote monitors for remote users. Similarly, when a videotape is being played, the lights in the room will automatically dim.

Along the same line, Mozer built the Neural Network House, a house that uses neural networks to balance the environmental needs of a user against the costs of heating and lighting the house [20]. The house gradually learns occupants' patterns of heating and lighting control for various locations in the house and various times of day. It uses this knowledge to predict user needs and automatically controls the heating and lighting, while, at the same time, attempting to minimize the overall expense incurred from them.

Environmental control was also a goal in the Intelligent Room project [5]. The Intelligent Room uses automatically sensed context along with explicit user input to adapt applications or services. Example services include turning off the lights and turning on soothing music when an occupant lies down on a sofa, and retrieving weather or news information that are relevant to the current context.

Bobick et al. also built an aware environment called KidsRoom [2]. KidsRoom was an interactive narrative space for children. The elements of the narrative continued based on the implicitly sensed activities of the children. These activities included talking to and dancing with avatars displayed on the walls.

These four aware environments all share the same property: they use implicit sensing but ignore any uncertainty in the sensed data and its interpretations. If the environment takes an action on incorrectly sensed input, it is the occupant's responsibility to undo the incorrect action (if this is possible) and to try again. There is no explicit support for users to handle or correct uncertainty in the sensed data and its interpretations.

A number of architectures that facilitate the building of context-aware services, such as those found in aware environments, have been built [3,9,13,14,35,23]. Unfortunately,

as in the case of the aware environments, a simplifying assumption is made that the context being implicitly sensing is 100% certain. Context-aware services that are built on top of these architectures act on the provided context without any knowledge that the context is potentially uncertain.

There are some exceptions to this assumption about certainty. We will examine two context-aware services that illustrate how individual services have attempted to take uncertainty of sensed input into account. The first is the Remembrance Agent, a service that examines the user's location, identity of nearby individuals, and the current time and date to retrieve relevant information [22]. The interpretation of the sensed context into relevant information is uncertain here. Rather than displaying the information with the highest calculated relevance, the Remembrance Agent instead presents the user with a list of the most relevant pieces of information and the relevance factor for each. In this way, the user can choose what is most relevant to the current situation from a filtered set.

The second service we will discuss is Multimodal Maps, a map-based application for travel planning [4]. Users can determine the distances between locations, find the location of various sites and retrieve information on interesting sites using a combination of direct manipulation, pen-based gestures, handwriting and speech input. When a user provides multimodal input to the application, the application uses multimodal fusion to increase the likelihood of recognizing the user's input. Rather than take action on the most likely input, if there is any uncertainty or ambiguity remaining after fusion, the application prompts the user for more information. By prompting the user for additional information, the system reduces the chance of making a mistake and performing an incorrect action.

QuickSet, another multi-modal map application also prompts the user for disambiguating information [6]. Rather than assuming that sensed input (and its interpretations) is perfect, these three services demonstrate techniques for allowing users to correct ambiguity in sensed input. Note that all three systems require explicit input on the part of the user before they can take any action. Our goal is to provide an architecture that supports a variety of techniques, ranging from implicit to explicit, that can be applied to context-aware services. By removing the simplifying assumption that all context is certain, we are attempting to facilitate the building of more realistic services.

8. FUTURE WORK

The extended Context Toolkit supports the building of more realistic context-aware services that are able to make use of ambiguous context. But, we have not yet addressed all the issues raised by this problem.

Because multiple components may subscribe to the same ambiguous events, mediation may actually occur simultaneously in these components. When one component successfully mediates the events, the other components need to be notified. We have already added the ability for input handlers to keep track of what is being mediated locally in order to inform mediators when they have been pre-empted. Although we have implemented this basic algorithm for handling multiple applications attempting to mediate simultaneously, we would like to add a more sophisticated priority system that will allow mediators to have control over the global mediation process.

An additional issue we need to further explore is how events from different interactions can be separated and handled. For example, in the In-Out Board service, it is assumed that only one user is mediating their occupancy status at any one time. If two people enter together, we need to determine which input event belongs to which user in order to keep the mediation processes separate.

We also plan to build more context-aware services using this new architecture and put them into extended use. This will lead to both a better understanding of how users deal with having to mediate their implicit input and a better understanding of the design heuristics involved in building these context-aware services.

Finally, this work does not attempt to answer the question of how best to handle mediation in such settings. The design of mediation for distributed multi-user settings and in settings with implicit input is still an open question. Our architecture makes it easy for programmers to experiment with mediation techniques and we hope it enables us to learn more about appropriate ways of handling mediation.

9. CONCLUSIONS

The extended Context Toolkit supports the building of realistic context-aware services, ones that deal with ambiguous context and allow users to mediate that context. When users are mobile in an aware environment, mediation is distributed over both space and time. The toolkit extends the original Context Toolkit and supports:

- Timely delivery and update of ambiguous events across an interactive distributed system via partial event graph delivery;
- Delayed storage of context once ambiguity is resolved;
- Delivery of ambiguous context to multiple applications that may or may not be able to support mediation;
- Pre-emption of mediation by another application or component;
- Applications or services in requesting that another application or service mediate; and,

- Distributed feedback about ambiguity to users in an aware environment via output services in context widgets.

We also introduced design heuristics that play a role in the building of distributed context-aware services:

- Applications should provide *redundant mediation techniques* to support more natural and smooth interactions;
- Applications should provide facilities for providing input and output that are *distributed both in space and time* to support input and feedback for mobile users; and,
- Interpretations of ambiguous context should have *carefully chosen defaults* to minimize user mediation.

We demonstrated and evaluated the use of the extended toolkit by modifying two example context-aware applications and by creating a new context-aware application. We showed that our architecture made it relatively simple to create more realistic context-aware applications that can handle ambiguous context and demonstrated the use of the design heuristics for creating these types of applications.

ACKNOWLEDGMENTS

We would like to thank our colleagues at Georgia Tech and UC-Berkeley who helped us build the applications described in this paper, who used the applications and who provided guidance on the design of the architecture.

REFERENCES

1. BELLOTTI, V. et al. 2002. Making sense of sensing systems: Five questions for designers and researchers. in *Proceedings of CHI 2002*, 415-422.
2. BOBICK, A. et *a*l. 1999. The KidsRoom: A perceptually-based interactive and immersive story environment. *PRESENCE 8*, 4, 367-391.
3. BROWN, P.J. 1996. The stick-e document: A framework for creating context-aware applications. In *Proceedings of Electronic Publishing* '96, 259-272.
4. CHEYER, A. AND JULIA, L. 1995. Multimodal maps: An agent-based approach. In *Proceedings of the International Conference on Cooperative Multimodal Communication '95*, 103-113.
5. COEN, M. 1999. The future of human-computer interaction or how I learned to stop worrying and love my intelligent room. *IEEE Intelligent Systems 14*, 2, 8-10.
6. COHEN, P.R. *et al*. 1997. QuickSet: Multimodal interaction for distributed applications. In *Proceedings Of Multimedia '97*, 31-40.
7. COOPERSTOCK, J., FELS, S., BUXTON, W. AND SMITH, K. 1997. Reactive environments: Throwing away your keyboard and mouse. *CACM 40*, 9, 65-73.
8. CUTREEL, E., CZERWINSKI, M. AND HORVITZ, E. 2001. Notification, disruption and memory: Effects of messaging interruptions on memory and performance. In *Proceedings of Interact '01*, 263-269.
9. DAVIES, N., WADE, S.P., FRIDAY, A. AND BLAIR, G.S. 1997. Limbo: A tuple space based platform for adaptive mobile applications. *In Proceedings of Conference on Open Distributed Processing /Distributed Platforms '97*.
10. DEY, A.K., MANKOFF, J., ABOWD, G.D. AND CARTER, S. 2002. Distributed mediation of ambiguous context in aware environments. In *Proceedings of UIST 2002*, 121-130.

11. DEY, A.K., SALBER, D. AND ABOWD, G.D. 2001. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction Journal 16, 24*, 97-166.
12. DEY, A.K. AND ABOWD, G.D. 2000. CybreMinder: A context-aware system for supporting reminders. *In Proceedings of the International Symposium on Handheld and Ubiquitous Computing*, 172-186.
13. HARTER, A., HOPPER, A., STEGGLES, P, WARD, A. AND WEBSTER, P. 1999. The anatomy of a context-aware application. In *Proceedings of Mobicom '99*, 59-68.
14. HULL, R., NEAVES, P. AND BEDFORD-ROBERTS, J. 1997. Towards situated computing. In *Proceedings of ISWC '97*, 146-153.
15. KORTUEM. G., SEGALL, Z. AND BAUER, M. 1998. Context-aware, adaptive wearable computers as remote interfaces to 'intelligent' environments. In *Proceedings of the International Symposium on Wearable Computers*, 58-65.
16. LESHER, G.W., MOULTON, B.J. AND HIGGINBOTHAM, J. Techniques for augmenting scanning communication. *Augmentative and Alternative Communication 14*, 81-101.
17. MANKOFF, J., ABOWD, G.D. AND HUDSON, S.E. 2000. OOPS: A Toolkit Supporting Mediation Techniques for Resolving Ambiguity in Recognition-Based Interfaces. *Computers and Graphics 24*, 6, 819-834.
18. MCKINLAY, A., BEATTIE, W., ARNOTT, J.L. AND HINE, N. 1995. Augmentative and alternative communication: The role of broadband telecommunications. *IEEE Transactions on Rehabilitation Engineering 3, 3,* 254-260.
19. MORAN, T.P and DOURISH, P. 2001. eds. Special Issue on Context-Aware Computing. *Human-Computer Interaction Journal 16, 2-4*, 87-420.
20. MOZER, M. C. 1998. The neural network house: An environment that adapts to its inhabitants. In *Proceedings of the AAAI Spring Symposium on Intelligent Environments*, 110-114.
21. MYERS, B.A. AND KOSBIE, D.S. 1997. Reusable hierarchical command objects. In *Proceedings of CHI '97*, 260-267.
22. RHODES, B. 1997. The Wearable Remembrance Agent: A system for augmented memory. *Personal Technologies 1*, 1, 218-224.
23. SCHILIT, W.N., 1995. System architecture for context-aware mobile computing, Ph.D. Thesis, Columbia University, May 1995.
24. WEISER, M. 1991. The computer for the 21st century. *Scientific American 265*, 3, 66-75.